Visualizing Object Oriented Software: Towards a Point of Reference for Developing Tools for Industry

Mariam Sensalire and Patrick Ogao
Faculty of Computing and IT
Makerere University
Kampala, Uganda
{msensalire,ogao}@cit.mak.ac.ug

Abstract

Developing a software visualization tool that gets high acceptability in the industry or research community would imply success for that particular tool. In the past, many tools have been developed within the academic arena with many more currently being developed. The rate of commercial success for the developed tools however does not match their development rate. In this paper the views of expert programmers are sought on what should be incorporated in a software visualization tool. These views are sought after exposing the programmers to three tools and allowing them to use the tools for a period of time. The results from the observations show that many of the desires of the expert programmers are not catered for in the currently existing tools. The potential need for a point of reference for developing tools for Industry is also discussed.

1. Introduction

Developers of software systems put a lot of effort in understanding the software they are working on especially if they were not part of the team that designed it [2, 8].Many times however, the developers can only rely on the source code as the true representation of the program being worked on [13, 20].

Due to the complexity of source code, visualization has been used to increase program understanding and many tools have in effect been developed to respond to this demand [2, 16]. Plenty of techniques have been introduced with the many tools developed [2, 10] however, it is still a challenge to choose the best one among them. In the absence of a guide, new researchers and developers of tools may choose to design based on what they think is useful.

Many times though, software developers do not involve the final users as much as they should during the development process [8, 22]. This is inclusive of those that develop software visualization tools. This nature of designing visualization tools under-utilizes the perceptual and cognitive theories leading to tools that are not properly evaluated by the users for whom they are intended [16, 22]. As such many of the designed tools are not used in practice. To be able to design a software visualization tool that is effective however, a proper needs assessment of the target users may need to be carried out [11].

Sim [18] noticed this challenge and advocated for benchmarking in an effort to better utilize the resources available in the software engineering field.

The work presented in this paper builds onto Sims work and addresses a different perspective of the benchmark. Emphasis here is put on the potential need for a point of reference for developing tools for Industry.

The rest of the sections in this paper are organized as follows; Section 2 looks at previous work that is related to what is presented in this paper while in section 3 the experiment that was carried out is presented. In section four, the results from the experiments are shown and finally section five concludes and provides areas for future research.

2. Previous Work

In the past, comparison of software visualization tools, developing tools based on user needs as well as benchmarking have all been looked at by different researchers.

Notable in the tool comparison area is Pacione [13] who compared five dynamic visualization tools and evaluated them with one user in relation to comprehension questions. Pacione noted that identifying the views that were appropriate for particular comprehension tasks was still a challenge. He also stated that the usefulness of the multifaceted three-dimensional model in software comprehension was yet to be evaluated.

There is also a body of work that advocates for the devel-

opment of tools based on the needs of the users since in the past, users of software visualization tools stated that their desired features were not always available in the tools.

Bassil [3] identified the need to carry out a survey that is targeted towards a more specific audience so as to get specific results for that group.This can then be followed by rapid prototyping which can help in reducing the time and effort in designing a tool which may not be effective [22, 14]. Likewise, Cordy [5] used his experience in industry to observe that it is an oversight to assume that a method that works well in academia would automatically be successful in industry.

3. The experiment

This section looks at the results of observing five expert programmers using three visualization tools. Their likes and queries with the tools used are discussed in an effort to show what the experts require if they are to adopt tool support.

3.1 Tools

Three tools were used for the study. These were Code Crawler ([15], [9]), Creole ([4], [12]), and Source navigator [19]. These tools were carefully selected based on the criteria of being freely available, the ability to visualize object oriented software and the difference between the techniques used during visualization.

Selection of tools that utilize different techniques was very important in order to avoid results based on only a given method of visualizing.

Code Crawler: This is a tool that combines the traditional visualization graph display with simple metrics about the software [6]. Code crawler is incorporated within Moose which is an extensible language independent environment for re-engineering object oriented systems [7].

Creole: Creole is an Eclipse plug-in that integrates Shrimp [21] with the Eclipse platform's Java Development Tools (JDT) [4]. Its main purpose is to provide both the high level program views as well as different dependencies within the source code .It is able to display code using radial, grid, and tree map techniques.

Source Navigator: This is a source code analysis tool [19] which enables code editing, display of classes and their relationships as well as call trees. It uses a very simplistic display format and does not use a lot of color and animation.

3.2 Source Code

Three different sets of source code were used for the three different tools. Code Crawler was evaluated using the Lucene search engine source code, Creole was tested using Apache beehive code whereas Source Navigator was tried using Apache tomcat code.

A different subject system was assigned for each tool in order to ensure that there was no familiarity with the code for each session. Using the same code could bias the participants as knowledge from the previous session could be carried forward.

Lucene Source Code Lucene is an information retrieval Application Programming Interface (API), originally implemented in Java by Doug Cutting. It is free, open source and released under the Apache Software License [1].

Apache beehive Apache Beehive is a Java Application Framework designed to make the development of Java Enterprise Edition (EE) based applications quicker and easier [1]. It was contained in 17 general packages with an average of 7 sub-packages within each main package.

Apache Tomcat Code Apache Tomcat is a web container that implements the servlet and the Java Server Pages (JSP) specifications from Sun Microsystems. It provides an environment for Java code to run in cooperation with a web server [1].

3.3 Participants

Five expert programmers from Industry participated in the study. They were all male with over ten years experience both in programming and computer usage. They were experienced with the object oriented paradigm with knowledge of at least two object oriented languages The three projects chosen were all in java - a language that all the participants had working knowledge of.

3.4 Procedure

The five participants were all exposed to the three tools one tool at a time. Each participant was given a 5 minutes introduction to a tool. After which they had 5 extra minutes for familiarizing with the tools themselves or seek any extra information. During this stage, sample tasks were given for each tool in order to prepare for the actual experiment.

After the familiarization stage, 2 tasks for each tool were given to the participants, one task at a time. The tasks given out were;

i Describe the static structure of the system, ie the main classes and their relationships.

ii What would be the effect of deleting a particular class or method from the source code?

Those tasks were replicated for all the three tools however the second task was modified according to the source code being analyzed.

i For Code crawler the second task was What would be the effect of deleting the Hook class?

ii For Creole, the second task was changed to What would be the effect of deleting the org.apache.beehive.netui.tags

iii Source Navigator's second task was naming the effect of deleting the DbStoreTest class.

These tasks were chosen because they could test the tools capacity to provide answers concerning the large scale and small scale static make up of the source code. The second task was able to test further the relationships and external references of given parts of the code. This was necessary as a typical programmer would need both [13]. The answers to these tasks were recorded on a pre-arranged answer sheet that covered all the three tools.

After working with a particular tool, there was a 2-minute break before proceeding to the next tool. This duration was kept short so as to ensure that the participants attention was not diverted form the experiments. At the end of the tasks for all the three experiments, a questionnaire was filled followed by an interview before terminating the session.

4. Observations

It was observed that more time than had been planned was taken during the experiment. The the pre-study questionnaires used the allocated time, however the observations, the answers during the tool exposure and the post study interview and questionnaire all took more time than had been allocated. During the course of the experiment, the participants made a lot of comments about the tools and also asked for extra functionality which was not always supported. The results from the questionnaires interviews as well as those observations are combined to provide the generalized results. This experiment was not aimed at choosing the best tool among the three observed. The results will therefore not be organized on a tool-by-tool basis.

5. Results

The results of the study showed that there was a lot that was desired by the programmers but was found to be lacking. The main queries observed are noted below:

5.1. IDE Integration

There was concern of integrating the visualization tools with an IDE. The reasoning was that when one visualizes, it is usually for a purpose. If the desire is to add more code to existing software, then it would be too much effort to switch between the visualization tool and the environment that is being used to program. So even if a tool is able to generate amazing visualizations, the effort and time spent switching between the two environments may have an effect on the knowledge preservation for programming thus the desire for integration. Creole is noted to have achieved this well especially if the programmers development environment is eclipse.

5.2. Fast responses

The speed of achieving the visualization was also highly complained about. Having programmed for a while, code was not as difficult for the experts to comprehend as it is for the novices. This meant that the switch to tool support had to be supported by the speed of achieving the respective task. More than two users stopped the generation of call graphs because they felt it was taking too long and a third user said that they would have achieved better results even with a plain IDE as opposed to waiting for a solution that takes too long. So even after the visualization has been generated but the timing is not right, it is not considered useful. Source Navigator was appreciated by all the participants for its speed of response.

5.3. Minimal effort

There was preference for minimal effort during the generation of the visualization. Having to export and import code in another format was found to be cumbersome for the participants. As much as possible direct code manipulation was prefered and in cases where that was not possible, close proximity to code was encouraged.

5.4. Good search abilities

The ability to search the visualization was another desired component that the users felt could have been addressed better. Even a good visualization that can not be manipulated was not found to be useful. Searching was desired both at the displayed diagram level and at the source code level in a manner that was related. Creole covered this element to a greater extent.

5.5. Clear display

Simple clear displays were prefered by the participants. If for example a class was shown, the participants desired

its name to be shown as well in close proximity to the class. The named class hierarchy of code crawler was especially mentioned as having been very clear.

6. Limitations

There are limitations to the approach used in this work that may hinder the level of generality of the results. The experiments carried out were in a controlled environment, a method that has been criticized in the past [14, 17]. The area of static visualization was also especially looked at despite the fact that a combination of static and dynamic visualization may be useful for object oriented software. The lack of a guideline for task selection during tool evaluation was also a limitation.

6.1. Conclusions and Future work

From the results of the study carried out combined with literature from previous studies, it can be concluded that better visualization tools and techniques can be achieved if the views of the target group are sought before hand . In order to develop tools that can be accepted in industry, concentration should be put on finding out the needs of the industry first and then later developing to meet those needs. Future work will involve improving the experiment and applying it to a plain IDE so as to compare results.

References

[1] ASF. www.apache.org/. *The Apache Software Foundation (ASF)*, 2007.

[2] M. Balzer and A. Noack. Software landscapes: Visualizing the structure of large software systems. *Joint EUROGRAPHICS-IEEE TVGD Symposium on Visualization*, 2004.

[3] S. Bassil and R. Keller. Software visualization tools: Survey and analysis. *Ninth International Workshop on Program Comprehension*, page 4, 2001.

[4] CHISEL. http://www.thechiselgroup.org/?q=creole. *University of Victoria, Canada*, 2006.

[5] J. Cordy. Comprehending reality practical barriers to industrial adoption of software maintenance automation. *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 196, 2003.

[6] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering approach combining metrics and program visualization. *Proc. of the Working Conference on Reverse Engineering (WCRE'99), IEEE*, 1999.

[7] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, page 13, 2000.

[8] R. Elferink, D. Griffiths, and S. Zondergeld. Comparing software development models: Structural problems in the cathedral and bazaar metaphors. *7th IMAC Conference on Localization and Globalization in Technology Design, Use and Transfer as a Subject of Engineering Education, Duisburg-Essen, Germany*, 2004.

[9] M. Lanza. Codecrawler-lessons learned in building a software visualization tool. *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pages 409–418, 2003.

[10] J. Maletic, J. Leigh, A. Marcus, and G. Dunlap. Visualizing object oriented software in virtual reality. *Proceedings of International Workshop on Program Comprehension (IWPC01)*, pages 21–13, 2001.

[11] A. Marcus, L. Feng, and J. Maletic. 3d representations for software visualization. *Proceedings of the 2003 ACM symposium on Software visualization, San Diego, California*, page 27, 2003.

[12] R. Michaud, M. Storey, and X. Wu. Plugging-in visualization: Experiences integrating a visualization tool with eclipse,. *ACM Symp. on Software Visualization, (Softvis'2003),*, 2003.

[13] M. Pacione, M. Roper, and M. Wood. A comparative evaluation of dynamic visualization tools. *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE),Victoria, BC,Los Alamitos, pp. 80-89, CA: IEEE CS Press,*, pages 80–89, 2003.

[14] C. Plaisant. The challenge of information visualization evaluation. *Proceedings of the working conference on Advanced visual interfaces*, pages 109–116, 2004.

[15] SCG. www.iam.unibe.ch/scg/research/moose/index.html. *Software Composition Group - University of Berne, Switzerland*, 2006.

[16] T. Schafer and M. Mezini. Towards more flexibility in software visualization tools. *VISSOFT*, pages 64–69, 2005.

[17] B. Shneiderman and C. Plaisant. Strategies for evaluating information visualization tools: Multi-dimensional in-depth long-term case studies. *Advanced Visual Interfaces Conference Venice Italy*, 2006.

[18] S. Sim, S. Easterbrook, and R. Holt. Using benchmarking to advance research:a challenge to software engineering. *Twenty-fifth International Conference on Software Engineering, Portland, Oregon,*, pages 74–83, 2003.

[19] Source-Navigator-Team. http://sourcenav.sourceforge.net,. *GNU,*, 2006.

[20] M. Storey. On integrating visualization techniques for effective software explorations. *Proceedings of IEEE Symposium on Information Visualization*, 1997.

[21] M. Storey, K. Wong, E. Fracchiat, and H. Miillert. On integrating visualization techniques for effective software exploration. *Proceedings of IEEE Symposium on Information Visualization*, 1997.

[22] M. Tory and T. Moller. Human factors in visualization research. *IEEE Transactions on Visualization and Computer Graphics Vol 10, No.1*, 2004.